Author contact information

Robert E. Filman

Lockheed Martin Advanced Technology Center / O / 96-10 B / 254E
3251 Hanover Street
Palo Alto, California 94304–1191
415–354–5250
415-424-2999 (FAX)
internet: Filman@stc.lockheed.com

Keywords:
Software reengineering
Software renovation
Software sustainment
Modernizing legacy systems
Software maintenance

Applying AI to Software Renovation

Robert E. Filman
Lockheed Martin Palo Alto Research Laboratories
O / 96-10 B / 254E
3251 Hanover Street
Palo Alto, California 94304–1191
415–354–5250
Filman@stc.lockheed.com

Abstract

Lockheed Martin InVision provides software renovation and sustainment services, including analyzing systems for “interesting features,” transforming systems to new environments, and recasting systems to new architectures and languages. We seek an optimal blend of effort by automating the straightforward parts of a reengineering task under human control. We achieve this automation through a judicious combination of artificial intelligence and compiler-compiler techniques. This paper describes the InVision tool set and reengineering process and presents some examples of the applications of this technology.

1. Introduction

Every software manager knows better than to lightly discard an existing software system. Such systems not only form the backbone of existing enterprises, but also embody critical knowledge about the actual processes of an application or organization. Legacy software is an asset. On the other hand, new operating conditions, new markets and emerging technologies are tempting occasions for improvement. These opportunities motivate modernizing software from out-of-date hardware to contemporary processors; from ancient or obscure languages, databases, and interfaces to current standards; and from centralized, closed systems to modern distributed, enterprise-wide open architectures. Software managers face the problems of preserving existing software assets and ensuring the continuing smooth operation of the organization while nevertheless responding to new requirements and taking advantage of emerging capabilities.

Modernization possibilities include buying commercial, off-the-shelf (COTS) software to provide a particular function (and then integrating it into the existing environment), redeveloping a system from scratch, or renovating (reengineering) an existing system to conform to new requirements. Each of these has its advantages and disadvantages. COTS software, if available, affordable, and appropriate can be a straightforward modernization technique. However, rarely are all these criteria met. Redevelopment offers the opportunity to radically update functionality, with concomitant high expense and risk. Reengineering seeks to take advantage of the knowledge embodied in existing systems to economically and reliably achieve modernization. (In practice, most projects use a hybrid approach, employing appropriate COTS components when available, redeveloping systems that require radical change, and reworking the rest.)

Lockheed Martin InVision provides software reengineering services to commercial organizations, to the government and government contractors, and within Lockheed Martin. These services include analyzing systems for "interesting features," transforming systems to new environments, and recasting systems to new architectures and languages. InVision seeks an optimal blend of effort by automating the straightforward parts of reengineering under human guidance and control. (The computer system thus acts as an "apprentice" to the human expert.) This combination of human domain understanding with computer attention to detail produces an overall superior reengineering result. InVision achieves this optimum by (1) developing and acquiring a collection of tools for the flexible analysis and transformation of existing systems, and (2) providing these tools to engineers skilled in the tool operation and modification, and knowledgeable in the source-and-target languages and environments. Artificial intelligence techniques and mechanisms play a prominent role in InVision's tool set.

This paper describes the InVision tool set and some of the applications it has enabled. Section 2 describes some of the uses of software renovation techniques. In Section 3, we turn to describing our tools and their use. We have applied our tools to a variety of applications, spanning tasks in program analysis and transformation. Section 4 presents a more detailed discussion of several renovation projects and research efforts. We briefly compare this work with other software reengineering efforts in Section 5. We close with a few remarks on the nature of semi-automated software reengineering.

2. Uses of Software Renovation

Broadly, software renovation can be used for *analysis* and *transformation*. Analysis describes the state of an existing software system. This information can be applied to maintenance, migration, and/or later transformation. The output of analysis can range from a static set of text reports, to node and edge graphs and on through an interactive query/browsing tools. Transformation is the process of modifying an existing system to meet the demands of a changing environment. The output of transformation is the original program rephrased in an alternative dialect or language, hosted on a different operating system and / or interfaced to a different operating environment.

Examples of useful analyses include:

(1) **Cross references.** These include *calling-trees*—which program elements invoke which other program elements, and *set/use–reports*—which data locations are accessed or modified by which other components.

(2) **Data and control flow.** Data and control flow reports show the progression of data values or overall system control through a system. Uses of control and data flow information include establishing the business rules of a system, enabling restructuring of a system, anticipating the effect of changes to data structures and values, and identifying dead (unreachable) code.

(3) **Software metrics.** Software metrics can be generated that suggest the difficulty of maintaining a particular software component.

(4) **Standards violations.** Legacy systems were often developed without coding standards. Modern developers often fail to follow standards. Violations may include use of language constructs beyond the standard language (for example, custom extensions to FORTRAN or K&R C function declarations) or outside the stylistic standards of the organization (for example, GOTO statements or multiple returns from a subprogram). It is possible to analyze the program structure to determine standards violations. Frequently, such a system can suggest or implement changes to make the code compliant.

(5) **Inventory analysis.** Rarely can the owners of a large application system identify all the source components. Reporting mechanisms can identify missing, unreachable, or redundant modules.

(6) **System dependencies.** An important element of software reengineering is identifying those places where the code invokes system-dependent routines. Thus, in porting from one operating system to another, it may be critical to identify calls to the first operating system's native routines.

Typical applications of program transformation include:

(1) **Programming language translation.** This includes moving systems from a non-standard or moribund language (for example, Jovial or CMS2) to a modern standard (for example, Ada), from older dialects (for example, FORTRAN IV or COBOL68) to modern variants (for example, FORTRAN 77 or COBOL II), and from proprietary systems (for example, proprietary 4GLs) to an open environments like C and COBOL. Much of (but, for a quality result, by no means all) such transformations can be accomplished automatically.

(2) **Replatforming.** Often, the nominal language of a system remains the same, while other elements of its environment (for example, hardware, language dialect, compiler, database system, or graphic user interface) need to be changed.

(3) **Remodularization.** Even systems with sound architectures erode under the muddling force of maintenance, and not every system has had the benefit of an architecturally organized youth. By examining the interconnectivity of system components, it is possible to recommend more appropriate modularizations of those components. Remodularization can serve as a prelude to programming language translation, as an appropriate organization for one language may be a poor structure for another.

(4) **Generalization and reuse.** Programmers often create application code in a hurry. Near the end of a project or during maintenance, it is often more expedient to copy and edit an old routine than to modify the original to be more general. However, the result of compounding such behavior is many copies of "almost the same" code in a system. Similarly, for large systems separate component implementations often arise from a lack of global perspective. It is possible to recognize certain kinds of redundancy and produce a generalized, reusable version of redundant code, thereby reducing overall system size.

(5) **Uniformly insert behavior.** Transformation can be used to uniformly insert behaviors (for example, debugging or metering mechanisms) into a system.

Before transformation, a skilled reengineer must analyze the system and determine the most cost-effective transformation strategy. Software reengineering projects must strike a delicate balance between the cost of developing automated tools and the cost of performing a task manually. Additional automation is worthwhile only if its development costs can be amortized over the available reengineering tasks. For example, developing rules for converting function pointers in C to Ada83 requires substantial effort. (In the most general case, it may not be possible to create a faithful rendition of every C function pointer in Ada.) An appropriate strategy uses automation to trace through the C inventory to identify function pointers and their uses, but leaves to the software reengineer the choice of implementation mechanism in the translated program (for example, generics or case-statements).

3. Automation

Realistic reengineering efforts must be based on actually examining the system to be reengineered.[1] For that reason, we base our tools on reading code, job

[1] Though the original project may have created copious documentation, documentation lies. Because

control language, and data definition language (and other such program descriptions) into an internal representation, and then manipulating that internal form. A knowledgeable software reengineer directs and controls this process. It is best if the internal representation both reflects the structures and relationships of the program and is amenable to facile programmatic manipulation. Once in internal form, programs can be written to analyze and report on the program structures, and to transform these structures (for example, a representation of the same program in another language). These reports and transformed programs can then be examined or printed out for use by software reengineers or for inclusion in the final system. Figure 1 illustrates this basic process: (a) parsing to an internal representation, (b) annotating that representation with information inferred about the program, (c) manipulating and transforming that representation, and then (d) presenting the results of those manipulations and transformations.

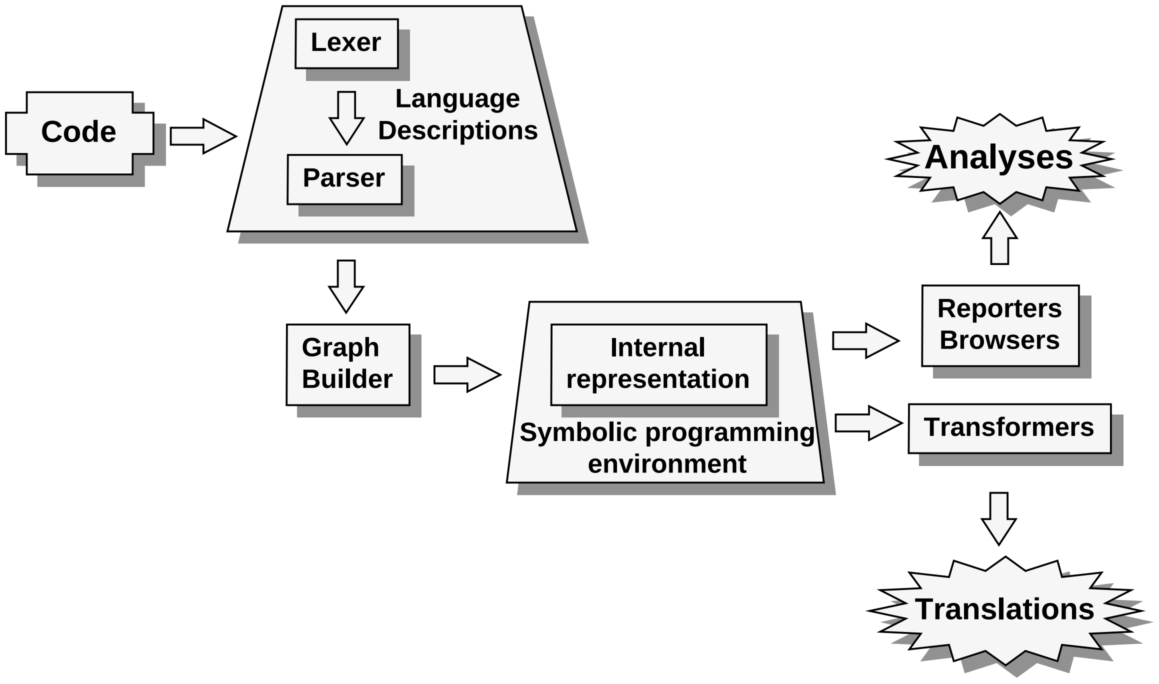


Figure 1: InVision system architecture

documentation effort has not tracked the system maintenance, and because the original documentation was incorrect or incomplete, old documentation is, at best, suspect. In practice, this is often a moot point; documentation for legacy systems is usually nonexistent.

Our primary tool environments have been Common Lisp (Steele, 1990) and Lisp enhanced with Refine (Reasoning Systems, 1990).[2] We have augmented those environments with both general and application-specific tools; we discuss some of those tools below. Technologically, we can be understood as combining compiler-compiler algorithms (e.g., compiler-compilers, regular-expression lexers, control and data flow) with artificial intelligence mechanisms (symbolic representation, objects and models, multi-dimensional objective functions and pattern-directed inference.)

3.1. Models and Grammars

In the introduction to this section, we discussed creating an internal representation of a program. The transition from program text to internal representation has two critical components:

(1) **The domain model.** A domain model is a framework for representing programs. We realize this framework as a set of class definitions. The slots of such classes encode the subparts of the program. Such a model recognizes the semantically common elements of the program, classifying them in an inheritance hierarchy. Thus, a language may be built of "sentences," including both "data sentences" (declarations) and "executable sentences" (statements). The executable sentences may divide into I/O sentences, iteration sentences, assignment sentences, and so forth. One seeks in this hierarchy *comprehensivity* (the entire language is described), *proximity* (similar conceptual elements share close common ancestors) and *utility* (operations can be described in relatively few places). As a concrete complexity measure of domain models, we consider CMS-2Y. CMS-2Y is an Algol-like language popular for U. S. Department of Defense applications. Our CMS-2Y domain model is eight levels deep, contains 308 classes and 215 "abstract-syntax tree defining" attributes. Our primary vehicle for expressing domain models has been the Refine object system. We have also explored expressing models in the Common Lisp Object System (CLOS).

(2) **The parser.** Program text needs to be recognized and encoded in the internal representation. This process has three critical steps: *lexing* (breaking the input into words), *parsing* (recognizing the sentences and parts of speech of the input and roughly assembling them into a program repre-

[2] Refine is an application development environment that extends Common Lisp with (1) a very high-level programming language that combines imperative, functional, logic and pattern-directed programming, (2) an object system, (3) a yacc-like system for defining the syntax of programming languages, automatically constructing LALR(1) parsers for those languages, and using those parsers to construct object-based abstract syntax trees of programs, and (4) a GUI-builder. Reasoning Systems also provides "language workbenches" (language models, parsers, and certain pre-defined analyses) for Ada, C, COBOL and FORTRAN.

sentation), and *fixup* (rearranging the results of parsing into a proper model, including establishing the declarative connection between program elements and restructuring odd organizations encumbered by particular parsing algorithms). The end-product of this process is the representation of the program as an *abstract-syntax tree* (AST). Figure 2 shows the AST representation of a parse of a (pseudo-language) conditional statement.

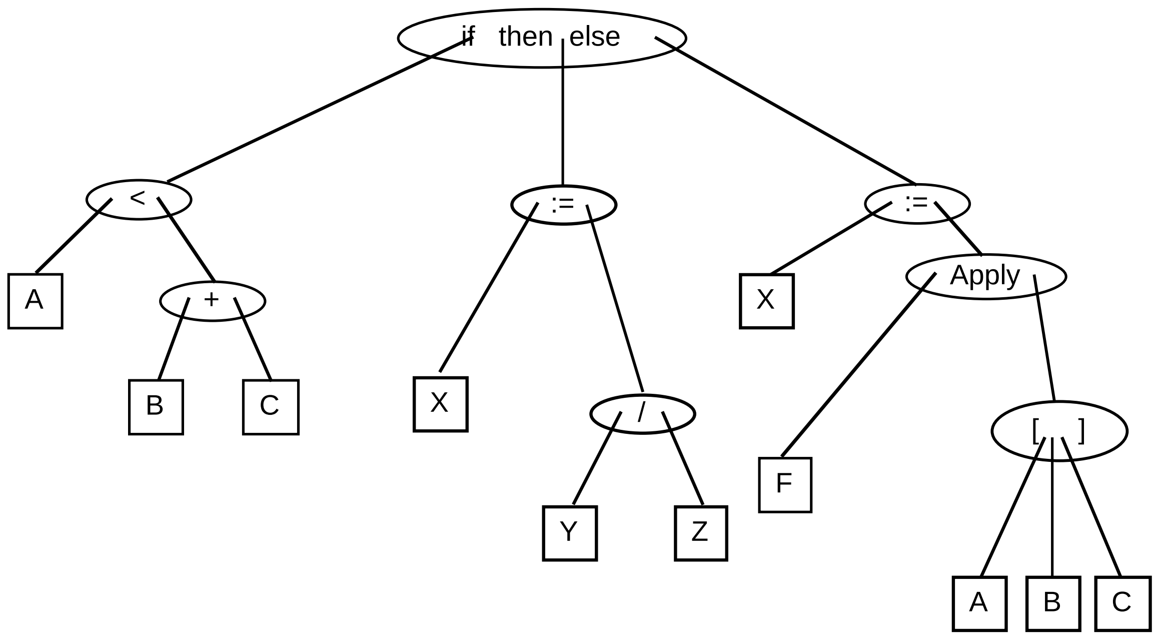


Figure 2: An abstract syntax tree representation of
"i**f** A < B + C **then** X := Y / Z **else** X := F (A, B, C)"

We have, by and large, used Refine for parsing and initial object construction, including both the predefined Refine workbenches (C, COBOL, FORTRAN, and Ada) and those additional workbenches we have constructed (for example, Jovial J3 and J73, CMS-2M and -2Y, InfoBasic, IMS-COBOL, IBM-370 assembler, IBM-370 JCL). Limitations in the size of system that can be accommodated by the "in memory" Refine environment have also led us to develop tools for transferring program representations between Refine objects and CLOS, and to storing program representations in a CLOS-based object-oriented database (Franz Allegrostore).

3.2. Annotation

We observe that the abstract syntax tree (AST) created by the parsing process approximates an appropriate internal representation for reengineering.[3] However, treating the nodes of the AST as simple record structures is too confining. Instead of tree nodes, a better representation creates *objects* that (1) can take additional annotation about inferred facts about the program, (2) can be grouped into classes and inheritance hierarchies to conveniently describe common properties, and (3) can be objects for object-oriented programming.

Since the grammar now defines both the syntactic structures of the language and its object classes and hierarchy, grammatical design for an object-oriented system requires additional care. Making the class hierarchically be semantically meaningful simplifies further processing: behaviors can be defined for all elements of a class.

Having represented the program as a network of objects, we can annotate those objects with additional information. Typical attributes include symbol-table references (for example, variable references pointing to their declarations), control-flow information (for example, statements pointing to those statements that can execute immediately after them), and data-flow information (for example, assignments pointing to those parameter and data values that went into their computation.) We call such an annotated structure an *abstract syntax graph* (ASG). As objects, the elements of the graph are amenable to object-oriented programming.

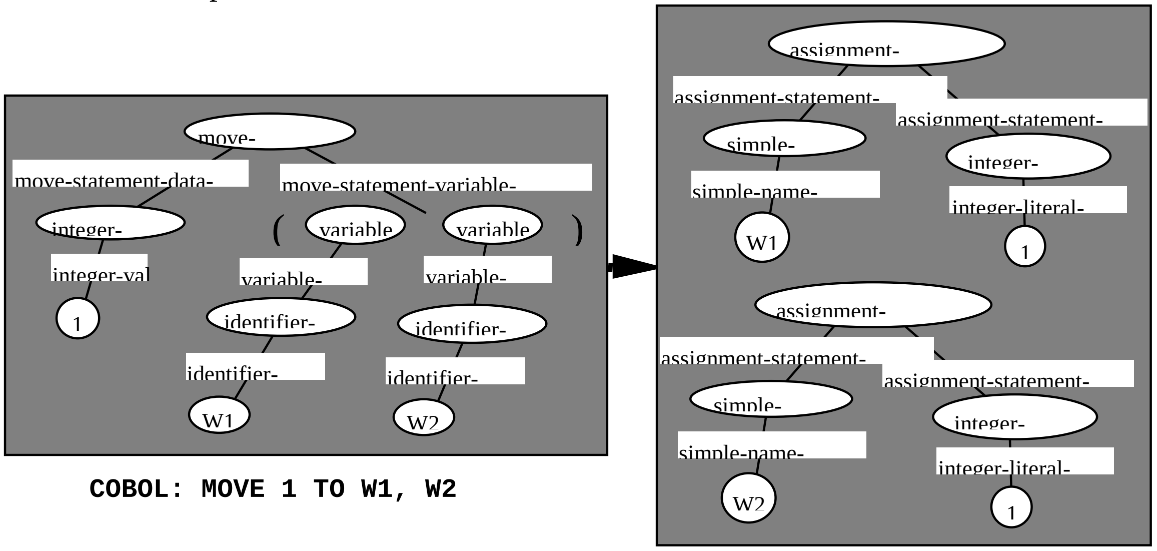
3.3. Analysis and Transformation

The two primary ways of manipulating ASGs are rule-based and object-oriented programming. Rules allow the straightforward description of relationships among (perhaps physically distant) elements and a similarly direct expression of what to do when discovering such relationships. However, rules are non-deterministic; pattern-matching can be obscure, making, rule-based systems can be difficult to debug. Object-oriented programming allows focusing behavior on individual kinds of elements, thereby providing the same interface to different classes of behaviors. On the other hand, it is more difficult to describe actions that relate a collection of typed objects with object-oriented programming. Each has its place in the reengineer's tool kit.

Analyses and transformations generate different kinds of results. Most commonly, analyses produce reports. A search of the ASG determines the data for

3 Sometimes, the best structure for parsing is not precisely the best structure for reengineering. For example, in many languages a function call with arguments is syntactically indistinguishable from an array reference. But for much of further processing (for example, variable set/use, and tracing subprogram calls) these are very different creatures. In such situations, it is useful to insert a transformational step, either immediately after parsing or after symbol table creation, that transforms the "readily parsed" representation to an "easily manipulated" one.

reports. Straightforward routines can convert these to either tabular or graphical form. Program transformation produces the textual representation of a program equivalent to the original in a different language or dialect. One way of accomplishing this is through object-oriented programming: having the objects of the ASG respond to messages telling them how to represent themselves in the new language (e.g., “Yo. You, COBOL if statement. Print yourself out in Ada.”) Alternatively, program transformation can create an AST in the target language. This AST can be made either by transforming the source ASG or by constructing a parallel AST. A grammar in the target language can then be "run backwards" to print this tree in the target language. (We have found it useful to annotate this target grammar with pretty-printing information, and have developed a pretty-printer for these annotations.) Figure 3 illustrates the abstract syntax tree transformations performed by a rule for transforming a COBOL multiple assignment statement to a sequence of Ada assignment statements. Figure 4 shows the text of a rule for making the implicit length of an assembly language instruction explicit. This rule uses the Refine grammatical pattern language, where fragments of textual programs are combined with pattern variables to express structure.



COBOL: MOVE 1 TO W1, W2

Ada: W1 := 1; W2 := 1;

Figure 3: Structural transformation from COBOL to Ada

```
rule make-length-explicit (x : defined-storage)
   x = ‘##r BALP @n @(typ) L @l < $(exas) >’
   & undefined? (l)
   & typ in [’C, ’P, ’Z, ’X, ’B]  % variable-length types
   & exas = [exa, ..]
   & len = bal-byte-length(typ, exa)
   --> x = ‘##r BALP @n @(’C) L @(len) < $([]) >’
```

Figure 4. A rule to make the implicit length of an assembler instruction explicit.

Building and using symbol tables provides an occasion to illustrate the use of object-oriented programming in reengineering. We first observe that many languages support some flavor of lexical scoping. This suggests organizing lexical contexts in a tree-structure, starting at some (global) root and creating child nodes at those places that allow new declarations. Finding an element in such a structure involves searching the current leaf node, and, if not found, recursively search up the tree towards the root. Languages may have other restrictions on symbols. Object-oriented techniques allow us to build lexical contexts that obey other disciplines. Second, languages typically manipulate multiple varieties of symbol tables. For example, a language may have one table for its variables, subprograms and type identifiers and another for statement labels. In such languages, it is perfectly legal to declare, say, a variable with the same name as a label. Usage (one doesn't “go to” a variable or increment a label) clarifies which variety of symbol is being referenced.

We use the term *lexical context* to refer to a collection of name spaces in a program. Given the above observation about different uses of the same name, any given lexical context may have a number of *symbol tables*, one for each name space in the programming language. We call an item in a symbol table a *symbol table entry*, or *s-t-e* for short. The fields of an s-t-e vary among languages. The three critical operations on lexical contexts are then:

(1) Given a symbol, a name space, and an s-t-e, add that <symbol, s-t-e> to the appropriate name space / symbol table.

(2) Given a symbol and a name space, lookup that symbol in the appropriate symbol table, returning the s-t-e, if any.

(3) Given a lexical context, create a child lexical context.

Object-oriented programming techniques allow us to declare a class of lexical contexts that implement these in a particular way (e.g., lexical contexts as a collection of symbol tables; symbol tables as hash arrays; lookup through sequential search up the lexical context tree). A different language specification might suggest a different organization for lexical contexts. Providing different method handlers permits the transparent use of alternative implementations by the rest of the system.

We note three interesting messages for the objects of the parse graph:

(1) What new s-t-e's do you have to contribute to the symbol table?

(2) What item of the symbol table do you use?

(3) Do you create a new (child) lexical context?

Most classes of parse objects respond negatively to these questions, but certain nodes (e.g., variable declarations, identifiers in expressions, and declare / begin / end blocks) require specialized behavior to, for example, create and return appropriate s-t-e’s. A generalized symbol table algorithm is then:

Given an object in the parse graph, *obj*, and a (current) lexical context, *context*:

(1) Send *obj* a “What new s-t-e's do you create?” message. The answer should be a set of <s-t-e, name-space> pairs. For each pair in that set,

  (1a) Mark the s-t-e it as being created by this object.

  (1b) Mark *obj* as having created this s-t-e.

  (1c) Send a message to *context*, asking to enter this s-t-e in the appropriate name space. For certain languages, it may be necessary for *context* to merge the information of this s-t-e with that of an existing s-t-e (as, for example, when multiple declarations build the symbolic definition of a name). “Global” or “external” declarations may also imply adding the information to the root symbol table, rather than the current leaf.

(2) Send *obj* a “What name do you use?” message. The answer should be either “none” or an <name, name space> pair. If a pair is returned,

  (2a) Ask *context* to look up this name in the appropriate name space, returning an s-t-e. [For certain languages, (languages that declare by use) this may imply creating and entering an s-t-e for this item.] Mark *obj* as using that s-t-e.

(3) Ask *obj* if it creates a new context. If so, create the context, noting the current context as its parent. Make that new context the current context.

(4) Apply this algorithm to each child of *obj*, using the current context

Rules and object-oriented programming provide a mechanism for taking a program in one language and converting it to a (usually semantically equivalent) program in another language. For all but the most trivial transformations, it has been our experience that purely automatic transformation does not produce maintainable output. Quality results require the interactive involvement of skilled software reengineers, both in the initial transformation process and in a *hand-polishing* post-transformation phase. Transformation, like many real AI applications, requires balancing a multi-valued utility function, making trade-offs between efficiency of execution, maintainability, faithfulness to the original source code, and fidelity to the style of the target language. Human reengineers can choose between alternative implementations, resolve difficult cases, and apply real-world knowledge to the reengineering process. The state of the art of transformation systems resembles not so much the expert system, able to solve a problem by itself as the expert’s apprentice, able to perform useful (though often low-level and boring) work under the guidance of the skilled master. This apprentice lacks the overall world view required for comprehensive performance. Presently, automation can ease the task of quality program translation, but is far from eliminating human involvement.

A concrete, low-level example may make this clearer. Given a variable declaration in the language C, we may need to transform that variable to Ada83. The simplest thing to do is to declare a corresponding variable in Ada, but that may not work. The C program may use the variable's address, illegal for stack-allocated simple Ada variables. This suggests implementing the variable as an Ada pointer to an object allocated from the heap (and changing all non-address uses of that variable to follow the pointer). This will always work, but introduces both computational and intellectual complexity. For example, if the variable is declared in a recursive subroutine, explicit allocation and deallocation will need to be done on the Ada variable. The deallocation becomes even more complex in the presence of exceptions, as exception handlers must be built that handle the deallocation. However, if the address of the variable is used only to achieve the effect of Ada **in out** and **out** parameters (a common cliché), a better resolution is to declare a simple variable and modify the corresponding subprogram declarations. If the variable's address is only occasionally used for true pointer purposes, then the software reengineer will need to trade off among maintainability, execution efficiency and semantic fidelity. These kinds of transformational decisions require both a system-wide perspective on the program and human skill and judgment.

4. Applications

We have applied our tool set and reengineering skills to several dozen application projects. These have included projects for migrating software between platforms, translating systems from one language to another, converting code to support different databases or user interfaces, and creating a variety of custom system analyses. Table 1 lists a selection of these projects. Figure 5 illustrates the overall process of a reengineering application. We note that each stage of the process produces useful output and that many applications do not require following the process to the end of the diagram.

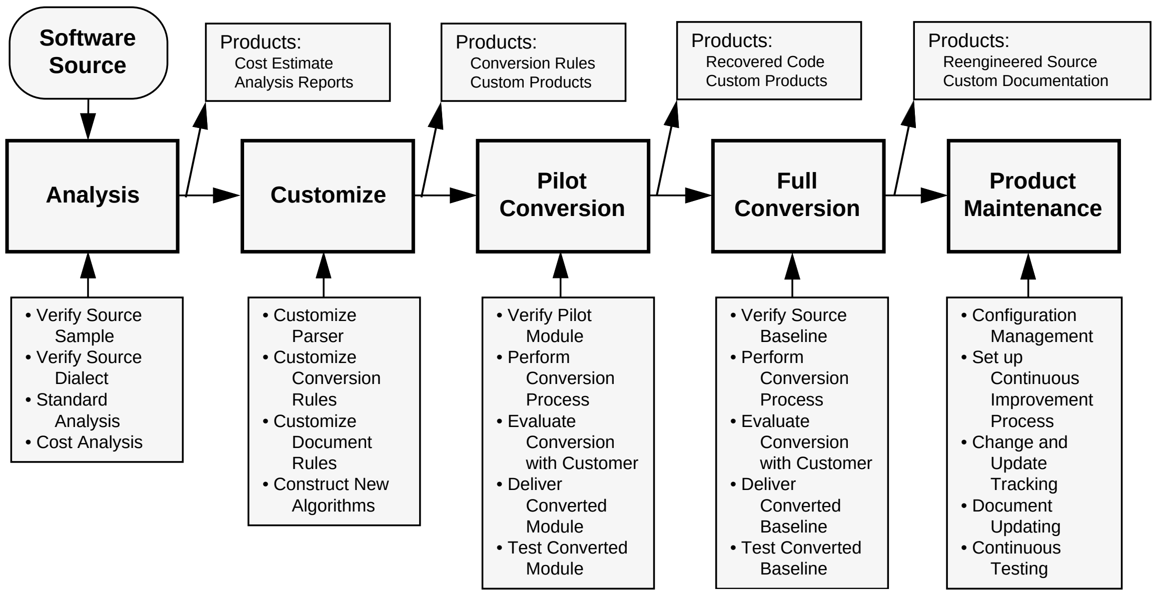

Software Source
Products: Cost Estimate Analysis Reports
Products: Conversion Rules Custom Products
Products: Recovered Code Custom Products
Products: Reengineered Source Custom Documentation
Analysis
Customize
Pilot Conversion
Full Conversion
Product Maintenance
• Verify Source Sample • Verify Source Dialect • Standard Analysis • Cost Analysis
• Customize Parser • Customize Conversion Rules • Customize Document Rules • Construct New Algorithms
• Verify Pilot Module • Perform Conversion Process • Evaluate Conversion with Customer • Deliver Converted Module • Test Converted Module
• Verify Source Baseline • Perform Conversion Process • Evaluate Conversion with Customer • Deliver Converted Baseline • Test Converted Baseline
• Configuration Management • Set up Continuous Improvement Process • Change and Update Tracking • Document Updating • Continuous Testing

Figure 5: The reengineering process

Of these application projects, we have selected five for detailed discussion: standards checks, platform migration, language translation, system restructuring and database conversion. These applications illustrate both the power and variety of reengineering tasks enabled by combining symbolic program manipulations with skilled software reengineers.

- Analyzed Ada for standards violations:
- Migrated Honeywell Fortran to generic Fortran
- Translated accounting application from COBOL to Ada
- Restructured C application
- Analyzed hierarchical DB for conversion to relational DB
- Simplified assembler to a wide-spectrum language
- Translated simulation application from C to Ada
- Translated proprietary DB system to IMS
- Restructured, Debugged, and fixed simulation software in VAX Fortran
- Analyzed COBOL RDB application with embedded DBMS code
- Analyzed embedded application VAX Fortran
- Analyzed Data General Fortran
- Translated systems software from C to Ada
- Migrated Honeywell Fortran to Silicon Graphics:
- Translated Data General Fortran to Standard C:
- Analyzed HP data processing Fortran
- Analyzed, restructured, and documented Fortran
- Restructured Jovial flight software
- Analyzed Unisys transaction-processing COBOL
- Generated documentation for C
- Analyzed flight-system CMS2-M
- Translated Jovial system to Ada

Table 1. Selection of InVision Software Reengineering Projects

4.1. Standards checks

Every sensible software development effort requires its programmers to conform to some coding standards. Standards help ensure code quality, prevent the use of language constructs that can produce erroneous results and encourage system development in a way that minimizes maintenance costs. Typical coding standards (for Ada) are:

- Do not compare floating-point numbers with the built-in equality operator.
- Out parameters should have values assigned to them on each and every logical path of the subprogram.
- Avoid anonymous types

- Eliminate unused object declarations (unused functions, procedures, and variables).
- Use **elsif** whenever possible to clarify logic
- Eliminate unused **with** statements

Given the ASG for an Ada program, one can detect such coding standards violations by describing each violation as a pattern or program that tests for the violation, and then traversing the ASG looking for violations. These tests can be as simple as checking, for each instance in the class of equality tests, that its arguments are not types derived from floats to as complex as processing each control-flow path in the program, looking for unassigned out parameters. (Of course, we are limited to checking structural properties of the code. We can easily recognize violations of a standard forbidding GOTO statements or requiring assignment to every out parameter on every path through a procedure. We can do little with a standard of the form, "Use meaningful variable names.")

We applied reengineering technology to detecting and reporting on project-specific coding standards violations for a large system. The project had 600K LOC (lines of code) of Ada and approximately 150K LOC of FORTRAN. Manual methods for auditing the code for standards conformance required 200 to 300 hours per 10K LOC module to detect violations of twelve Ada standards. Quality assurance for the project requested our support in reducing this manual effort. We customized our environment to perform the checks. This took approximately 200 hours (including requirements definition, design, implementation, and testing). The Lockheed Martin InVision reengineering tool set then performed the coding standards checking for all twelve violations at the rate of 15 minutes per 10K LOC module. Program Software Quality Assurance estimates that the tool saved $3 million in quality assurance auditing costs and provides at least a 1000-fold productivity improvement over manual methods.

4.2. Platform migration

Lockheed Martin InVision performed a FORTRAN platform migration task, taking FORTRAN 77 applications running on a Honeywell mainframe computer to FORTRAN 77 on a Silicon Graphics (SGI) workstation. A critical point about FORTRAN system migration is that no two FORTRAN dialects are identical. For example, every dialect seems to have its own syntactic representation of octal and hexadecimal constants, and some contain elements far beyond the ken of the FORTRAN standard [e.g., embedded, lexically scoped subroutines (GTE) or recursion (Cray)]. In the case of the Honeywell to SGI migration, Honeywell FORTRAN 77 contains constructs such as repeat statements, constant statements, and multiple assignment statements. (Respectively, these translate to do statements, parameter statements, and sequences of single assignment statements in SGI FORTRAN.) More critically, the word sizes of the two machines are different: the Honeywell is a 36-bit machine, while the SGI is 32.

An additional complication was the customer's creation of and use of a FORTRAN preprocessor, PREPP. PREPP supports higher-level constructs such as begin-end blocks, logical and arithmetic case statements, and different types of repeat blocks with associated next and break statements. Our second task was to transform these PREPP/Honeywell systems to FORTRAN 77. We preserved the maintainability of the translated systems by using comments and indentation to retain the structured appearance of the original code.

Lockheed Martin InVision created a set of customized tools to automate the task of translating Honeywell FORTRAN to Silicon Graphics FORTRAN and PREPP/Honeywell to a FORTRAN 77 dialect that is as platform independent as possible. This required customizing our tools to handle both Honeywell FORTRAN extensions and the PREPP preprocessor. We then transformed the resulting ASGs to express either the Honeywell extensions in SGI FORTRAN or the Honeywell/PREPP extensions in standard FORTRAN 77. For the Honeywell to SGI systems, the next step in the transformation process is to identify and convert the platform dependent aspects of the code. These include calls to system routines and code dependent on the machine word size. A PREPP/Honeywell translation requires the definition of many new statement labels. The final step in this type of translation is eliminating the unused statement labels and sequentially renumbering the remaining statement labels and their corresponding references.

The particular conversion involved the translation of 16 Honeywell FORTRAN systems to SGI FORTRAN 77, consisting of a total of 240K lines of code.

4.3. Database conversion

We are currently developing a tool to aid in the conversion of hierarchical IMS databases to relational models. In comparison with hierarchical systems, relational databases are more flexible, often more efficient, and run on significantly less expensive hardware. We recognize, in our tool development, the critical role databases play in modern organizations, and the need to proceed cautiously in their conversion. Our first tool efforts are therefore devoted to understanding the organization of IMS database systems and suggesting translations, not their automatic conversion.

Database reengineering involves three phases: translating the database schema, converting the actual data, and reengineering the application code. (Commercial tools are available to support, to some extent, the first two of these. The last is the hardest, as the relational model may compel significant algorithmic revision of programs. The problem is compounded by the fact that organizations often have millions of lines of database applications.)

Schema conversion. Straightforward conversion of IMS schema to relational models suggests mapping IMS segments to relations and links to foreign keys. However, such a conversion leads to inefficient applications—the system may be expending considerable energy on preserving the order of a collection of records when no actual code is dependent on that order. Considering the actual access patterns in the application code can produce better results. Our basic conversion approach associates a relational table with each physical segment type. Its columns hold the segment data and, (where appropriate) the physical and logical parents. Pointers between segments are represented using foreign keys. The mapping from fields to columns is complicated by IMS overlapping and non-covering fields. Our resolution is to partition the data into non-overlapping segments and to express actions in terms of multiple columns.

Date conversion. The primary difficulty in converting the actual data is the necessity of maintaining the proper logical parent links. The actual physical conversion can be either through a program that calls both the IMS and relational databases, or by encoding the IMS data in SQL statements or flat files and then reading that data in appropriately on the target machine.

Application conversion. The most complex part of the conversion process is converting the actual code. The translation process relies on data- and control-flow analysis, literal propagation, cliché recognition in programming patterns, and report generation. Recognition of IMS call patterns is needed because it is not possible, or at least extremely inefficient, to convert individual IMS calls in isolation. Rather, we detect certain access patterns which can be efficiently translated as a group. Part of the difficulty in translating IMS calls in isolation is because IMS database applications contain explicit control structures to iterate through all segments matching a given query. Arbitrary statement sequences can be mixed in with the database iteration, obscuring the actual operation. In a relational database, a single query retrieves all matching tuples. Cursors are used to traverse the query result. Similarly, one may have to write an explicit IMS loop to perform unitary relational operation, such as summing the values of a field of a set of predicate-satisfying records. We have written programs to recognize such clichés.

We use control flow analysis to build a control flow graph of IMS database calls. Often, thousands of lines of COBOL can be reduced to less than a page of information. Even lacking further automation, such an analysis is helpful in maintaining the system code.

Data flow analysis determines where a particular data item is used and to propagates static values. Determining the actual data used allows simplifying the SQL query to ask for only that data.

Another IMS feature allows an application to sequentially traverse all data of a database. Such traversals cannot directly be translated into SQL as each

SQL query can only return data of a single type. Polymorphic traversals need to be analyzed manually to determine a proper relational translation.

InVision applied the first versions of these tools in the analysis and modernization of a COBOL / IMS inventory application composed of approximately 500,000 lines of application code and 100 megabytes of data. This system and its application are described more completely by Polak, *et. al.* (Polak, 1995)

4.4. Restructuring C

Software system restructuring is the process of reorganizing the architecture of a software system to accommodate a shift in the underlying design principles. C to Ada translation illustrates the need for performing software system restructuring. C systems consist of a set of independent functions and variables. Generally, structured design principles guide the creation of C systems. On the other hand, Ada is an object-based language with support for encapsulation, information hiding, generic packages and subprograms. The goal of producing maintainable Ada code requires us to restructure to-be-translated C systems to organizations more natural for Ada.

InVision performs software system restructuring by automated tools supported by manual processes (Chu, 1992). The restructuring tool accepts a C system (consisting of a collection of C files) as input and parses the C code to produce an ASG. A semantic component then traverses the graph and collects primitive structural and semantic information. This information includes: (a) all the functions in the C system, (b) the calling relationships between these functions, (c) the global variables of the system, and (d) the set-use relationships between the functions and these variables.

The overall strategy of the restructuring tool is heuristic and opportunistic. It travels up the calling hierarchy of the C system. As it moves from some function B to function A (when A calls B), the restructuring algorithm attempts to include B as part of A in an abstract component. If B is called exclusively by A, then B can become part of A. On the other hand, if B is called by some other function C or shares global data with some function D (and hence has other dependencies) then two choices exist: (1) B and everything it depends on (the global data and D) can be considered to be a part of A, or (2) B can be made a part of C. This basic principle is applied recursively to build ever larger, more “abstract“ software components.

We applied our software restructuring tool on two projects. The first consisted of 113K LOC comprising of 139 C functions. The system successfully restructured the system into 15 independent components (roughly the same as subsystems), and 8 abstract data types. We used the system to restructure the C application into six hierarchical Ada packages with four nested Ada packages. The original C developers reviewed the output of the restructuring process and indicated that the new structure clustered related functions and variables into conceptually meaningful Ada packages. The second system consisted of 13K LOC and consisted of 177 C functions. In this system we located 7 abstract data types and restructured the system into 15 Ada packages with 12 nested sub-packages.

4.5. Language translation

InVision has performed several language translation projects, including conversion of C, COBOL, Jovial, and CMS-2 to Ada; FORTRAN to C; and FORTRAN to PL/1. Here we discuss the translation of a 50K LOC COBOL UNISYS OS-1100 logistics system into 35K LOC, functionally equivalent Ada system (Gray, 1995). The application was a collection of 32 independent programs melded into a distributed system over several dozen sites. Difficulties of conversion included the use of two different communication systems, two databases, operating systems calls, overlays, and unstructured and repetitious code.

We converted this system into 35K LOC of Ada, replacing the GOTO's with structured constructs, paragraphs and performs with procedures and functions, introducing strong typing, and generally producing a system that (save for some name choices) appears to have been developed originally in Ada. The process also revealed (and fixed) several bugs in the original systems and introduced a few algorithmic improvements.

This result was accomplished through a combination of automated and manual efforts. We created a number of transformations for taking COBOL constructs into Ada and applied these to the ASG representations of our COBOL programs. In practice, it is not worthwhile to expend effort developing transformations for little-used or difficult constructs. The result of the automatic part of the conversion was thus a mixed COBOL–Ada program. Human skill was then applied to complete the translation and polish the results.

We are also currently working on a research effort to convert assembly language to high-level language (Morris, 1996). Our efforts to date have produced a converter that has automatically reduces a certain assembly language programs to a wide-spectrum intermediate-language. Typically, the converted programs are 30-40% of the size of the originals. The conversion process recognizes clichés such as scanf, performs algebraic simplifications, resolves condition codes into real conditionals, discards assembler-specific "noise," and introduces structured constructs such as loops in place of GOTO's. This translation process relies on techniques such as control and data flow analysis and pattern-matching. We are continuing to work on this technology to expand the class of simplifications it provides and the quality of output it produces.

5. Related work

Software reengineering is where software engineering was twenty years ago: an emerging field, with the beginnings of theory, tools and methodology. Software reengineering draws heavily on its intellectual parents: software engineering, programming languages, and processing modeling, with a bit of an artificial intelligence influence. This paper has been about experience at applying automation to reengineering, automating various parts of the reengineering process, and the limits of automation. (Here we echo Yu (Yu, 1991), who argues that human expertise is critical for any but the most trivial reverse engineering task.) Automation of reengineering has been a fertile field of research, though the "Holy Grails" of automated program translation and understanding remain as elusive as translation and understanding remain for natural language systems. In this section we touch upon other's research related to the work described above.

The most straightforward use of automation in software reengineering is for systems to (generically) aid in the understanding of existing software. Most work on program understanding as been at either a lexical (token or string matching) or syntactic (parsing) level. Relational code analyzers (e.g., The Dependency Analysis Toolset (Wilde, 1989), MasterScope (Teitelman, 1981), Cscope (Steffen, 1985), AQL (Paul, 1994), and the C Information Abstraction System (Chen, 1990; Grass, 1992)) deduce some code-level structural relationships (e.g., who calls who, who sets who) from source code and provide a queryable database of these relations. The application of Refine technology in this respect is illustrated by Boeing's maintenance of an on-line browser of a 200K LOC COBOL application (Newcomb, 1995a). Similar structural ideas are shown in the work of Cross and Hendrix in visualizing the control structures of Ada programs concurrently with their source (Cross, 1995), the MITRE assembly language workbench (Roberts, 1994), and Andersen's COBOL/SRE system to aid in understanding COBOL systems (Ning, 1994).

While relational code analyzers capture the syntactic structure of systems, they do not present a higher-level, conceptual view of code, systems, and architectures. The AI approach to greater system understanding is to attempt to recognize patterns in code (Kozaczynski, 1992; Kozaczynski, 1994; Wills, 1990; Rich, 1990; Letovsky, 1988; Ning, 1993; Ning, 1994, Quilici, 1994). While such systems can deal with code, they lack attachment to the modeled domain. This theme has been explored by Biggerstaff et al. (Biggerstaff, 1994) who argue that parsing-based technology lends itself to recognizing programming-oriented concepts (e.g., algorithms), but that application-oriented concepts (the coupling of the program to the domain) are inaccessible without human intervention. The synthesis of these approaches is seen in the work of Quilici and Chin (Quilici, 1995), where a stereotypical design element recognition phase is followed by interactive graphical display and queries.

The particularly applications we have considered also have parallels in the literature. Refine-based quality assurance systems include the work of Wells et al. (Wells, 1995) for Fortran and C, and, generically, the standards mechanism built into Refine (Markosian, 1994a).

Database reverse engineering has been a fertile field with important economic implications. Hainaut et al. have develop an methodology of database reverse engineering and tools to support this methodology (Hainaut, 1993; Hainaut, 1995). They discuss the difficulty of extracting conceptual database descriptions from the messiness of non-toy code, and describe a transformational approach to extracting conceptual information from real programs. The Reverse Engineering in CASE Technology method (RECAST) takes COBOL source and deduces Structured Systems Analysis and Design Method (SSADM) documentation (Edwards, 1993; Edwards, 1995). Yang and Bennett described a tool that (like our IMS work) considers actual code analysis in deriving entity-relation models from COBOL code (Yang, 1995). Jarzabek and Keam describe a reverse engineering tool for semi-automatic, knowledge-based and incremental reverse engineering of software, for example, extracting data models from COBOL flat files (Jarzabek, 1995). Their works recognizes that the particular design abstractions needed to understand a program (e.g., control flow graphs, ASTs, calling graphs) are a function of the goals of a reengineering project. They also describe an SQL-like query language for browsing program structure.

More popular than simple restructuring is the current attempt to recognize the “objects” in conventional code. Choi and Scacchi (Choi 1990), Hutchens and Basili (Hutchins, 1985) and Maarrek (Maarrek, 1988) search for subprograms grouped by data or calls. Yeh et al. (Yeh, 1995) work from AST to infer abstract data types structurally from C code. Newcomb and Kotik (Newcomb, 1995b) apply a variety of structural and flow analysis techniques for object extraction of COBOL programs. Gall and Klösch (Gall, 95) combines structural analysis with domain data analysis to discover objects. Markosian et al. (Markosian, 1994b) describe their experience with Refine in developing tools to reverse engineer COBOL applications, including the database elements.

Automatic translation from legacy languages to high-level, portable systems remains an elusive goal. Bennett et al. (Bennett, 1992) describe the Maintainer’s Assistant, which employs formal program transformation techniques in support of maintenance activities. Like our work on assembler, Bennett et al. report on a system applied to the analysis of IBM 370 assembler by transformation into a wide-spectrum intermediate language. Andrews et al. describe work on maintaining non-syntactic, macro structures in doing a source-to-source translation from a proprietary Algol-like language to C (Andrews, 1996). One larger-scale experiment on automated translation was performed by the Naval Surface Warfare Center, Dahlgren Division (NSWCDD) (Samuel, 1995) which compared three automatic translators for CMS-2 to Ada (Cohn, 1991; Lock, 1994; Sampson, 1994). Within the context of legacy code modernization, Samuel and his coauthors make a telling point: translation systems, while somewhat useful, also required “A great deal of clean up to the produced code.” They add, “This finding illustrated the fact that not all CMS-2 code can or should be translated,” a simultaneous recognition of both the necessity of hybrid modernization techniques and the difficulty of translation. NSWCDD continues its efforts to improve translation technology.

6. Closing remarks

We have described InVision, its tool set, and some of the applications enabled by these tools. Our description has included a process of program manipulation (parsing, structural interconnection, transformation and analysis, reporting and printing) and software renovation. This process is relies on AI technology and techniques. We have argued that this is an appropriate architecture for automating many of the tasks of software reengineering. We have also presented examples of where even clever automated tools cannot produce a high-quality reengineering result. In these situations, human intelligence is required to examine the particulars of the situation and select or create new structures accordingly.

Acknowledgments

This paper describes the work of not only the author, but of the entire InVision group: Bruce Amberden, Tim Bickmore, Larry Chavez, Jackie Combs, William Chu, Kalyan Dutta, David Ecsery, Ron Gray, Al Mendall, Paul Morris, Les Nelson, Lori Ogata, Sukesh Patel, Wolf Polak, Joe Sullivan, Scott Williams, and David Zimmerman. This paper is a revision of (Filman, 1995) and includes material from references (Filman, 1994), (Gray, 1995) and (Polak, 1995).

References

Andrews, K., Del Vigna, P. and Molloy, M. 1996. Macro and File Structure Preservation in Source-to-source Translation. *Software—Practice and Experience* 26 (3) pp. 281–292.

Bennett, K, Bull, T. and Yang, H. 1992. A Transformation System for Maintenance—Turning Theory into Practice. *Proc. 1992 IEEE Conference on Software Maintenance*, pp. 146–155.

Biggerstaff, T. Mitbander, B. G and Webster, D. E. 1994. Program Understanding and the Concept Assignment Problem. *Communications of the ACM*, *37* (5) pp. 72–83.

Chen, Y. F., Nishimoto, M. Y and Ramamoorthy, C. V. 1990. The C Information Abstraction System. *IEEE Transactions on Software Engineering 16* (3) pp. 324–334.

Choi, S. C and Scacchi, W. 1990. Extracting and Restructuring the Design of Large Systems. *IEEE Software, 7* (1) pp. 66–71.

Chu, W. C and Patel, S. 1992. Software Restructuring by Enforcing Localization and Information Hiding. *Proc. 1992 IEEE Conf. on Software Maintenance*, pp. 165–173.

Cohn, R. A., Kossiakoff, A. and Noble, J. C. 1991. *CMS-2 to Ada Translation Tools*, Technical Report FS-91-148, Fleet Systems Department, The John Hopkins University, Applied Physics Laboratory.

Cross, J. H. and Hendrix, T. D. 1995. Using Generalized Markup and SGML for Reverse Engineering Graphical Representations of Software, *Proc. Second Working Conference on Reverse Engineering,* pp. 3–6.

Devanbu, P. T. 1992. GENOA—A Customizable, Language- and Front-End Independent Code Analyzer. *Proc. Fourteenth International Conference on Software Engineering,* pp. 307–319.

Edwards, H. M. and Munro, M. 1993. RECAST: Reverse Engineering from COBOL to SSADM Specifications. *Proc. Fifteenth International Conference on Software Engineering.* pp. 499–508.

Edwards, H. M. and Munro, M. 1995. Deriving a Logical Data Model for a System Using the RECAST Method, *Proc. Second Working Conference on Reverse Engineering,* pp. 126–135.

Filman, R. E. 1995. Applying AI to Software Renovation, *Working Notes Third Workshop on AI and Software Engineering,* IJCAI-95, pp. 28–37.

Filman, R. E., Chavez, L. A and Patel, S. 1994. The Truth is in the Code, but it Takes a Human to Understand it: The Lockheed PRISM Software Reengineering Effort. *Proc. Sixth Annual Software Technology Conference.*, CD-ROM recs. 1499–1783.

Gall, H. and Klösch, R. 1995. Finding Objects in Procedural Programs: An Alternative Approach. *Proc. Second Working Conference on Reverse Engineering,* pp. 208–216.

Grass, J. E. 1992. Object-Oriented Design Archaeology with CIA++. *Computing Systems: The Journal of the USENIX Association 5* (1), pp. 5–67.

Gray, R., Bickmore, T and Williams, S. 1995. Reengineering COBOL Systems to Ada. *Proc. Seventh Annual Software Technology Conference..*

Hainaut, J. L., Chandelon, M., Tonneau, C and Joris, M, 1993. Contribution to a Theory of Database Reverse Engineering. *Proc. First Working Conference on Reverse Engineering,* pp. 161–170.

Hainaut, J.-L., Englebert, V., Henrard, J., Hick, J.-M and Roland, D. 1995. Requirements for Information System Reverse Engineering Support. *Proc. Second Working Conference on Reverse Engineering,* pp. 136–145.

Hutchens, D. H. and Basili, V. R. 1985. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering 11* (8) pp. 749–757.

Jarzabek, S. and Keam, T. P. 1995. Design of a Generic Reverse Engineering Assistant Tool, *Proc. Second Working Conference on Reverse Engineering* pp. 61–70.

Kozaczynski, V. and Ning, J. Q. 1994. Automated Program Understanding by Concept Recognition. *Automated Software Engineering 1* (1) pp. 61–78.

Kozaczynski, V., Ning, J. Q and Engberts, A. 1992. Program concept recognition and transformation. *Transactions on Software Engineering 18* (12) pp. 1065–1075.

Lock, E. 1994. Reengineering Concurrency Related Operating Systems Calls. *Fourth Reengineering Forum*, pp. 60-1–60-6.

Letovsky, S. 1988. *Plan Analysis of Programs*. Ph.D. thesis, Yale University.

Maarrek, Y. S. 1988. On the Use of Cluster Analysis for Assisting Maintenance of Large Software Systems. *Proc. 3rd IEEE Israel Conference on Computer Systems and Software Engineering*, pp. 178–186.

Markosian, L., Brand, R. and Kotik, G. 1994a. *Proc. Fourth Systems Reengineering Technology Workshop*, pp. 248–255.

Markosian, L., Newcomb, P., Brand, R., Burson, S., Kitzmiller, T. 1994b. Using and Enabling Technology to Reengineer Legacy Systems. *Communications of the ACM*, 37 (5) pp. 58–71.

Morris, P and Filman, R., 1996. "Mandrake: A Tool for Reverse-Engineering IBM Assembly Code," submitted.

Newcomb, P. 1995a. Legacy System Cataloging Facility. *Proc. Second Working Conference on Reverse Engineering*, pp. 52–60.

Newcomb, P. and Kotik, G. 1995b. Reengineering Procedural into Object-Oriented Systems. *Proc. Second Working Conference on Reverse Engineering*, pp. 237–249.

Ning, J. Q., Engberts, A. Kozaczynski, W. 1993. Recovering Reusable Components form Legacy Systems by Program Segmentation. *Proc. First Working Conference on Reverse Engineering*, pp. 64–72.

Ning, J. Q., Engberts, A and Kozaczynski, W. 1994. Legacy Code Understanding. *Communications of the ACM*, 37 (5) pp. 50–70.

Paul, S. and Prakash, A. 1994. Querying Source Code Using an Algebraic Query Language. *Proc. 1994 IEEE Conference on Software Maintenance*, pp. 127–136.

Polak, W., Bickmore, T and Nelson, L. 1995. Reengineering IMS databases to Relational Systems. *Proc. Seventh Annual Software Technology Conference*, CD-ROM.

Quilici, A. 1994. A Memory-Based Approach to Recognizing Programming Plans. *Communications of the ACM 37* (5) pp. 84–93.

Quilici, A and Chin, D. N. 1995. DECODE: A Cooperative Environment for Reverse-Engineering Legacy Software, *Proc. Second Working Conference on Reverse Engineering*, pp. 156–165.

Reasoning Systems, 1990. *REFINE User's Guide, Version 3.0*. Palo Alto: Reasoning Systems Inc.

Rich, C. and Waters, R. C. 1990. *The Programmer's Apprentice*. Addison Wesley, Reading MA.

Roberts, S. N., Holtzblatt, L. J. and Reubenstein, H. B. 1994. Reverse Engineering Assembly Language Programs. *Proc. 4th Reengineering Forum,* pp. 61-1–61-10.

Sampson, C. H. 1994. Translating CMS-2 to Ada. *Proc. Fourth Systems Reengineering Technology Workshop,* pp. 143–156.

Samuel, A. L., Sam, E., Haney, J., Welch, L., Lynch, J., Moffit, T. and Wright, W. 1995. Application of a Reengineering Methodology to Two AEGIS Weapon System Modules: A Case Study in Progress. *Proc. Fifth Systems Reengineering Technology Workshop,* pp. 69–79.

Steele, G. L., 1990. *Common Lisp: The Language, 2nd Ed.* Bedford, MA: Digital Press.

Steffen, J. L. 1985. Interactive examination of a C program with Cscope. *Proc. USENIX Association.*

Teitelman, W. and Masinter, L. 1981. The Interlisp Programming Environment, *Computer 14* (4) pp. 25–34.

Wells, C. H., Brand, R. and Markosian, L. 1995. Customized Tools for Software Quality Assurance and Reengineering, *Proc. Second Working Conference on Reverse Engineering,* pp. 71–77.

Wilde, N., Huitt, R. and Huitt, S. 1989 Dependency Analysis Tools: Reusable Components for Software Maintenance. *Proc. 1989 IEEE Conference on Software Maintenance,* pp. 126–131.

Wills, L. 1990. Automated Program Recognition: A Feasibility Demonstration. *Artificial Intelligence 45* (1-2) pp. 113–168.

Yang, H. and Bennett K. 1995. Acquisition of ERA Models from Data Intensive Code. *Proc. 1995 International Conference on Software Maintenance,* pp. 116–123.

Yeh, A. S., Harris, D. R. and Reubenstein, H. B. 1995. Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language. *Proc. Second Working Conference on Reverse Engineering,* pp. 227–236.

Yu, B. 1991. Large Software System Maintenance. *Proc. 6th Annual Knowledge-Based Software Engineering Conference,* pp. 171–177.